Procedia Computer Science

Volume 80, 2016, Pages 1450–1460

ICCS 2016. The International Conference on Computational Science

KGEN: A Python Tool for Automated Fortran Kernel Generation and Verification

Youngsung Kim[1], John Dennis[1], Christopher Kerr, Raghu Raj Prasanna Kumar[1], Amogh Simha[2], Allison Baker[1], and Sheri Mickelson[1]

[1] National Center for Atmospheric Research, Boulder, Colorado, U.S.A.
[2] University of Colorado, Boulder, Colorado, U.S.A.

Abstract

Computational kernels, which are small pieces of software that selectively capture the characteristics of larger applications, have been used successfully for decades. Kernels allow for the testing of a compiler's ability to optimize code, performance of future hardware and reproducing compiler bugs. Unfortunately they can be rather time consuming to create and do not always accurately represent the full complexity of large scientific applications. Furthermore, expert knowledge is often required to create such kernels. In this paper, we present a Python-based tool that greatly simplifies the generation of computational kernels from Fortran based applications. Our tool automatically extracts partial source code of a larger Fortran application into a stand-alone executable kernel. Additionally, our tool also generates state data necessary for proper execution and verification of the extracted kernel. We have utilized our tool to extract more than thirty computational kernels from a million-line climate simulation model. Our extracted kernels have been used for a variety of purposes including: code modernization, identification of limitations in compiler optimizations, numerical algorithm debugging, compiler bug reporting, and for procurement benchmarking.

Keywords: Kernel, Code Extraction, Mini-app, Python, Source-to-source Transformation

1 Introduction

There are many reasons why the application software stack needs to be studied including: rewriting the software to obtain improvements in performance or identifying the reasons for failure of the application. These types of studies have typically been performed using the complete application. However, use of the complete application induces significant overhead both in terms of execution time and resource consumption. These overheads could be significantly reduced if a kernel could be used that captures the computational characteristics of the original application. Janssen [10] defined a compute kernel as a self-contained program that embodies essential performance characteristics of the node-level aspects of an application. Janssen used kernels to "enable rapid exploration of new languages and algorithms" for co-design.

 Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2016

doi:10.1016/j.procs.2016.05.466

While compute kernels have become increasingly popular, creating these kernels is frequently time-consuming and laborious work. To create kernels it is necessary to create a kernel with the same computational characteristics, and provide it with the necessary input-data to drive kernel execution and output-data to verify correctness of the solution. The total amount of work necessary to create computation kernel is sufficiently large as to limit their broad use in the scientific software community.

In this paper, we introduce a new Python-based open source tool, Kernel GENerator (KGEN) that simplifies the creation of computational kernels [11]. KGEN both extracts the necessary Fortran code from the original application to create a standalone executable and instruments the existing application source code to capture input and output state data that can be used to verify the extracted kernel. In Section 2 we describe concepts of kernel extraction on a simple example program, and the internal calculations that KGEN performs. In Section 3 we describe the KGEN user interface. KGEN has been used to extract a significant number of kernels from the Community Earth System Model (CESM) [9] an earth system model whose development has been centered at the National Center for Atmospheric Research (NCAR). Section 4 describes how kernels generated by KGEN have been utilized and shared amongst a diverse group of software developers to perform performance optimizations on different computer architectures. Finally a description of related work in provided in Section 5, followed by conclusions and future work.

2 Computational kernel extraction

In this section, we describe the context of kernel extraction used in this paper and ideas behind our kernel extraction tool, KGEN. To motivate our discussion, we will use a simple program illustrated in Figure 1. There are three source files in this program. We want to extract the subroutine call to *calc* on line 26 of Figure 1 and turn this "call-site" into a kernel. A search of the source for the file "update_mod.F90" indicates that *calc* subroutine is defined on lines 40 to 47 of "calc_mod.F90 in the "calc_mod" module. An additional search of the source file "calc_mod.F90" reveals other subroutines that are used by the *calc* subroutine. A similar search must take place for each of the variables used within the *calc* subroutine, in this case "i", "j" and "output". Once all of the names of variables and subroutines have been resolved, it is now possible to create a kernel. Additionally, to generate state data, the original source code is instrumented to collect the input and output state through the addition of Fortran write statements. A new driver source file is created that calls the *calc* subroutine with the correct input state and to compare the output of the *calc* subroutine with the expected output.

Given the simplicity of the program in Figure 1, a kernel could be easily created without automation in a very short amount of time. However we are fundamentally not interested in extracting kernels for simple programs as shown in Figure 1, but rather from very large and complex codes like CESM that contains over 1 million lines of code. Kernel extraction from large complex applications is only possible by automating the kernel extraction process. In the next section we describe the necessary calculations to perform automated kernel extraction.

2.1 Automating Kernel Extraction

The tasks of kernel extraction can be divided into two tasks: 1) identifying a minimal subset of application source code that is necessary in order to construct a stand-alone executable and 2) generating the input and output state necessary to execute and verify the extracted kernel.

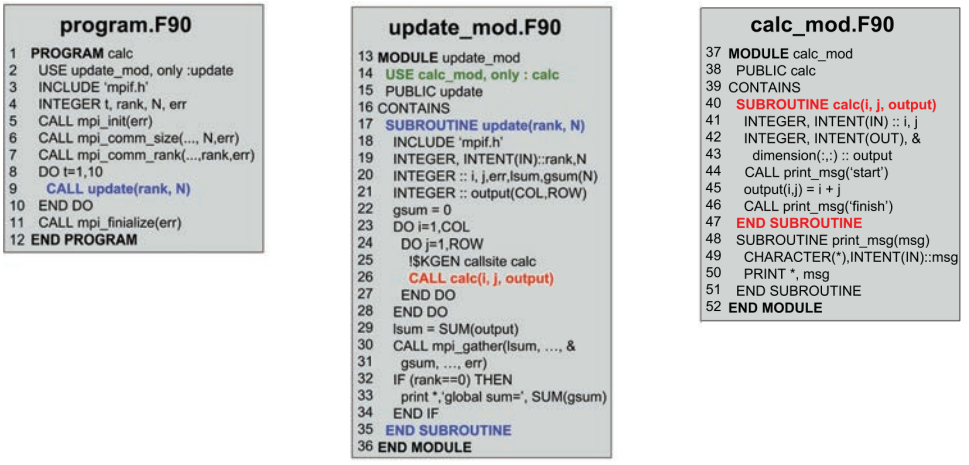

Figure 1: A simple example program to illustrate kernel extraction. The kernel to be extracted is a block of lines between 40 and 47 of "calc_mod.F90," while kernel call-site is on a line 26 of "update_mod.F90"

Identifying source lines for kernel extraction involves a search through the program source to resolve all variables used within the target subroutine. For example, a Fortran subroutine that is selected for extraction contains one or more statements. If one of these statements contains a variable, the search would begin by identifying a Fortran type declaration statement for the variable. If the type declaration statement uses additional variables or user defined data-structures, then these type declarations must be located as well. Such a search will typically involve multiple source files and multiple name-spaces. For example the *calc* subroutine was defined in *calc_mod* module or namespace. Abstract syntax tree (AST) representations for each source files are used for the search. The search continues until all necessary variables in all statements of the target subroutine are resolved.

During the search on the ASTs, it is possible to collect information about how variables are used within the kernel. For example, the information can be collected from the Fortran intent attributes of IN, OUT, or INOUT, or by analyzing if variables are used on either the left or right side of the equal sign in Fortran assignment statement. The content of "input" variables are saved before call to the target Fortran subroutine and "output" variables, which are used for verification of the extracted kernel, are saved after the subroutine call. If there are global variables used or assigned in the kernel, all the global variables should also be saved in a similar manner to remove "side-effect."

2.2 Implementing Automated Kernel Extraction

In this section, we describe how the concepts described in Section 2.1, are implemented within KGEN. Each of the following steps is illustrated in Figure 2.

- **User Specification:** The first step is the specification of which code to extract. Minimally, users need to specify which subroutine that should be extracted through the use of directives. However, it is common to provide additional information such as directory paths to Fortran modules and macro definitions used in the source files. This step is marked as "A" in Figure 2

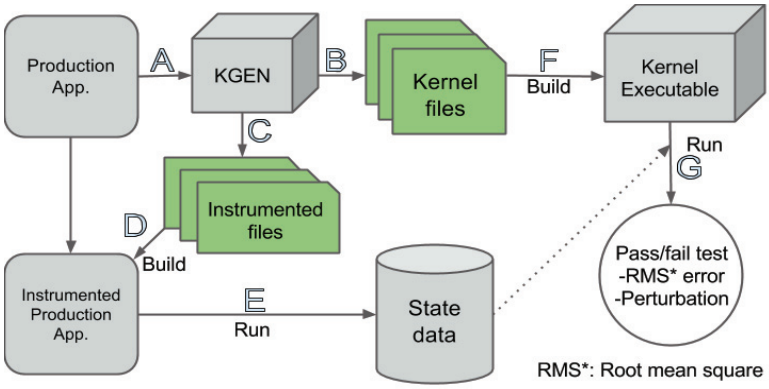

Figure 2: KGEN Workflow Diagram.

- **Generation of Kernel and Instrumentation Source Code:** KGEN generates two sets of Fortran output files. The Kernel fileset contain a minimal amount of code necessary to generate the standalone kernel executable and a Makefile. This step indicated by "B" in Figure 2. KGEN also generates an instrumented fileset. These files contain all the functionality of the original application source files with the addition of instrumentation to save state input and/or output data. This step is indicated by "C" in Figure 2.

- **Generation of Input and Output State:** The next step in the KGEN workflow is to re-compile the original application using the newly modified files. The instrumented application can now be run, which generates the files that contain the necessary state to drive and verify the standalone kernel. These steps are indicated as "D" and "E" respectively in Figure 2

- **Kernel Execution and Verification:** The final step in the automated kernel extraction process is to build and execute the standalone kernel. The build step is indicated by "F" in Figure 2, while the execution step is indicate by "G". The execution step "G" requires the input and output state generated in step "E" described above. Note that a verification capability, which currently includes a root mean squared error check, is also included in the kernel. The initial steps "A" through "E", which may require considerable knowledge about the original production application, are typically executed only once. Steps "F" and "G", which only require the ability to compile and run an executable, is typically repeated multiple times.

3 KGEN Workflow

In this section, we describe how KGEN is used in practice using the simple program in Figure 1 for illustrative purposes. In particular, we will step through the various phases described 2.2 starting with the user specification stage.

While KGEN is able to automate a great deal of program analysis, it does require a very modest amount of user intervention. At the minimum, the user must specify the location of the call-site that KGEN will use to generate the kernel. If the original application does not have a flat directory structure for its source code or it uses C preprocessor macros or variables, additional required inputs are required. Optional user intervention provides information to

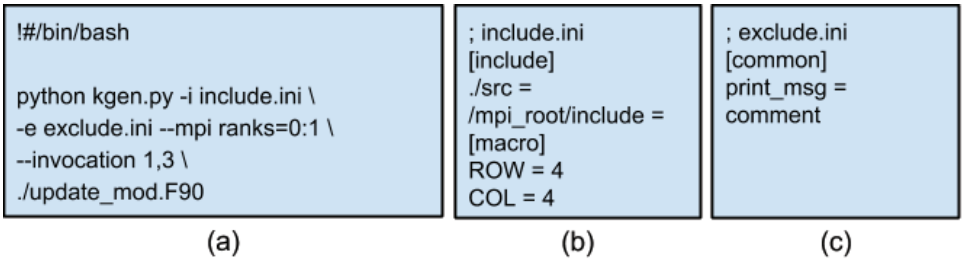

Figure 3: Panel (a) a shell script to run KGEN, panel (b) an include INI file to KGEN for providing macros and inclusion paths, and panel (c) is an exclude INI file another input file.

prune the call tree to simplify the analysis on the source codes, and to limit the amount of state data that is collected.

To specify the call-site to KGEN, the "callsite" directive is used just before the call-site in the original source file. The syntax of the directive is shown below.

```
!$kgen callsite callsite_name
```

Additional KGEN flags are illustrated in panel (a) of Figure 3 that limit the scope of collected data. Note that the call to subroutine *calc* is both nested in a loop and called from a parallel MPI program. KGEN provides flags to limit the capture of state information to particular MPI ranks and particular invocation of a subroutine. For example, "–mpi rank=0:1" flag limits the capture of state information from the *calc* subroutine to MPI ranks 0 and 1, while the flag "–invocation=1,3" indicates KGEN should only save data from the 1st and 3rd invocations to *calc.* Detailed specification information is passed to KGEN using the include and exclude INI configuration files. For example, the macro definitions and file search paths are provided using the include INI configuration file whose name is indicated by the "-i" flag. The contents of an include INI configuration file is shown in panel (b) of Figure 3. KGEN also provides the ability to prune the source tree. The pruning can greatly simplify the analysis that KGEN must perform and limit the resulting size of the extracted kernel. For example in Figure 1, the user may know that call to the subroutine *print_msg* is not relevant to the generation of the kernel. To exclude *print_msg* from kernel extraction, an entry is added in the INI configuration file indicated by the "-e" flag. The content of the exclude INI file is provided in panel (c) of Figure 3. The ability to prune is particularly important in the case where the application uses a large external scientific library like NetCDF [19], which may have a very deep call-tree.

Once KGEN has sufficient information to completely construct an AST for the target kernel, it generates two output filesets in two separate sub-directories. In the kernel sub-directory, the three source files that are generated when extracting the *calc* call-site from the simple program are shown in Figure 4. The "kernel_driver.f90" is an entry point for kernel execution that calls the parent of the call-site. On lines 110, 111, and 114 in Figure 4, three variables are read from a state data file. After the call to kernel at line 118, there are several additional code blocks for output verification and timing measurement. The "calc_mod.F90" file contains the computational kernel *calc*. Note that only code necessary to execute the kernel is included. In particular, the subroutine "print_msg" is commented out on line 23 and 25 in Figure 4.

In the state sub-directory, a modified version of the input file "update_mod.F90," which is illustrated in Figure 5, is generated. The original code structure is maintained while additional instrumentation code added to save input and output data. On lines 719 and 720 in Figure 5,

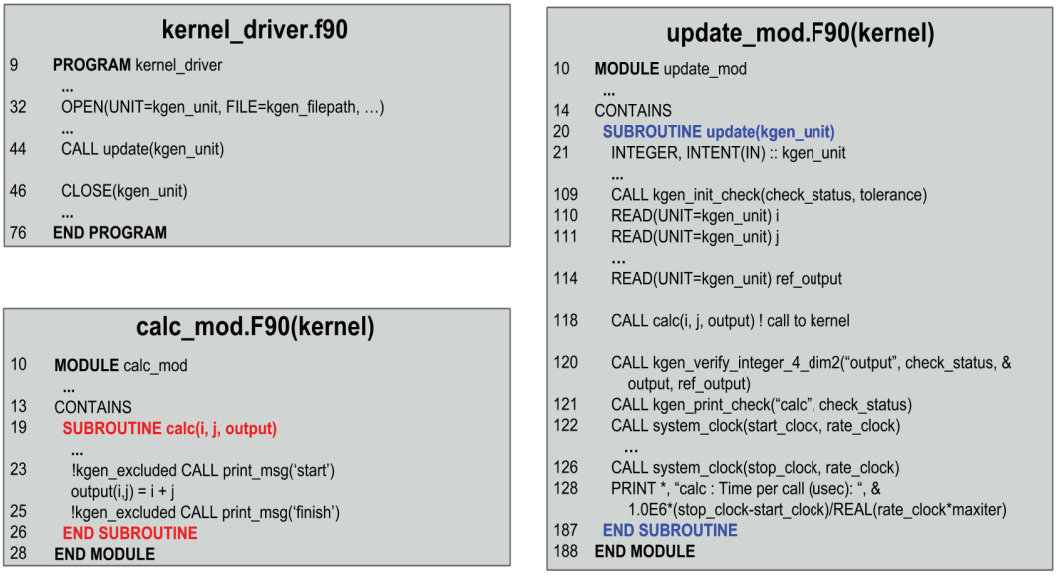

Figure 4: The kernel fileset that are extracted from the sample program in Figure 1.

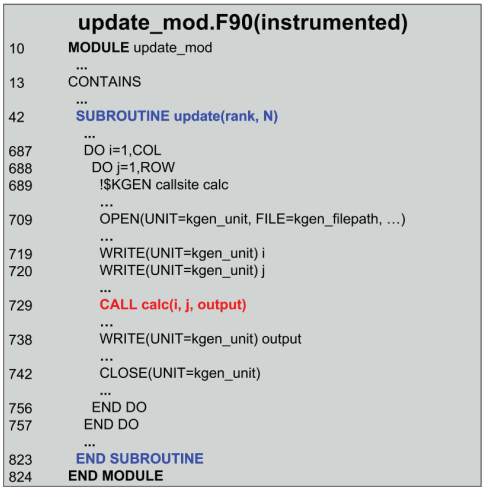

Figure 5: The instrumented fileset for generating state data.

two input variables, "i and j", are saved into an external file and on line 738, one output variable, "output," is saved after the call to *calc*. The original application is now re-compiled and executed using this instrumented "update_mod.F90". Once completed, the execution will generate the state data files in "kernel" sub-directory. With the state date files, the extracted kernel in "kernel" sub-directory can be built and executed. The execution of the kernel will output verification and performance results.

4 KGEN-generated Kernel Usage Examples

We next describe several examples of how KGEN-generated kernels have been used to increase programmer productivity. To date more than thirty kernels [12] were extracted from weather/climate simulation models including the Community Earth System Model (CESM) and the Model for Predictions Across Scales (MPAS) [20]. These kernels range in size from 100 lines to 6,000 of lines, and were extracted from several different component models. We focus on several kernels in particular including two versions of the Morrison Gettelman (MG) microphysics [18][6] written by atmospheric scientists at the National Center for Atmospheric Research (NCAR). The MG kernels are 3600 - 3800 source code lines in length and have approximately 80 output arguments. The Rapid Radiation Transport Model Global Climate Model version (RRTMG) was written by scientists at Atmospheric and Environmental Research a private company [4] and customized for inclusion into CESM. The RRTMG module which calculates long-wave radiation is 5600 lines in length and has 9 output arguments while the short-wave radiation module is 6700 lines in length and has 18 output arguments. We first describe how a MG1 kernel was used to identify a numerical instability. Next we describe how MG2 kernel was used to evaluate the effectiveness of compiler optimization and as a benchmark for a large scale high-performance computing procurement. Finally we describe how KGEN capabilities greatly simplify code modernization efforts for the longwave length and shortwave length radiation calculations in the RRTMG module.

4.1 CESM Ensemble Verification

CESM utilizes an ensemble based verification technique as a mean to provide software quality assurance. CESM-ECT [2] evaluates whether a new 1-year climate simulation (e.g., resulting from a different architecture, compiler, etc.) is consistent with an existing 150-member ensemble on a trusted hardware and software platform. While this ensemble verification approach allows for an easy identification of verification failure, it does not provide an easy method to identify the root-cause of the failure. Direct examination of the CESM code as a whole is prohibitive on many levels: CESM has more than one million lines of code, a one-year run at the target resolution on 900 cores requires several of hours of computer time and a potential long wait in a queue of a shared supercomputer. The pairing of KGEN and CESM-ECT offers an interesting opportunity to simplify root-cause analysis of a verification failure.

We focus on a verification result from CESM-ECT that indicated that simulation results from “Mira” an IBM Blue Gene/Q machine at the Argonne Leadership Computing Facility were statistically distinguishable from the accepted results from “Yellowstone” an IBM iDataplex Linux cluster at at the NCAR Wyoming Supercomputing Center (NWSC). Of the 120 variables that CESM-ECT tested, six ’suspicious” variables were identified , where four of which were identified by climate scientists as being influenced by the MG microphysics module. Utilizing KGEN, the MG kernel was extracted from CESM. Execution of the MG kernel on Mira also indicated a KGEN-based verification failure due to the presence of larger than expected normalized root mean squared errors for several of the output variables. After the addition of several print statements in the MG kernel, a problematic line of code that contained a fused multiply-add (FMA) was identified. When FMA was deactivated on Mira, the output of the MG1 kernel matched the trusted Yellowstone results. New ensemble members on Mira were generated with FMA deactivated that successfully passed the CESM-ECT verification tests. Use of KGEN greatly simplified the identification and elimination of the verification problems encountered on Mira. KGEN had successfully created a 3600-line program that was an accurate representation for the purposes of verification as the original one million-line application.

Instead of waiting in the queue for 3 days to run a several hour job, the KGEN kernel executed in 1000 μsec on a single core.

4.2 Performance Proxy

We next describe how KGEN was used to create a performance proxy. For decades small pieces of code have been used to test the performance of micro-processors and the compilers' ability to optimize code. While simplified synthetic benchmarks were initially used like Livermore loops [17], and STREAMS [16], much more complex performance proxies or Mini-apps [7][8] are now being used more extensively. A challenging aspect of creating a performance proxy is the need for it to be both simple enough to be accessible by a broad audience, and complex enough to be truly representative of challenging calculations. An additional challenge exists when creating performance proxies from rapidly evolving scientific applications. We have utilized KGEN to generate the MG2 kernel that has been utilized for multiple purposes. Once a version of the MG2 source code, which is an update to MG1 described in the previous section, was introduced in a release of CESM, we were able to very rapidly generate a standalone kernel using KGEN. The MG2 kernel was subsequently used as a benchmark for NCAR Wyoming Supercomputing Center (NWSC) procurement. MG2 was also released as a performance proxy to all interested Fortran compiler vendors. The motivation behind releasing MG2 to Fortran compiler vendors was to use it as a vehicle both to learn how to optimize non-trivial code, but also to understand limitations in the current optimizing compilers. MG2 was effective in achieving both goals.

4.3 Code Modernization

We next describe examples of how KGEN-generated kernels have been used to benefit code modernization efforts. CESM, a community model, which benefits from contributions for a large number of researchers, is constantly evolving. It is therefore critical that any code modernization effort be sufficiently rapid as to keep-up with the pace of scientific development. However, several factors make fast code modernization of CESM a challenging task: the cycle time of "modify-execute-evaluate" takes several hours to several days, there are few if any subroutines within CESM that consume a considerable percentage of the total time, there is a limited number of individuals having expert knowledge of both CESM and computer architecture. A strategy to overcome the difficulties on the code modernization is to split a large task into many smaller tasks that can be performed independently. Once the small tasks are defined, KGEN is used to generate computational kernels that can then be optimized by multiple engineers.

KGEN-based kernels of the MG calculations, described in the previous section as well as the long- and short-wave calculations in the RRTMG module have been optimized. In particular the execution time for the MG2 kernel was reduced from 1075 μsec to execute on a single 2.6 GHz Intel Sandybridge core when compiled with the Intel 15.0.1 compiler to 541μsec. The reduction in execution time was achieved by reducing the number of mathematical operations, using an internal GAMMA function, rewriting of the sedimentation sub-cycle loop and elimination of elemental routines, and rearranging a number of loops to enable vectorization. While a single developer primarily performed this optimization work, it benefited significantly from discussions with compiler developers. The use of the kernel allowed many discussions to occur with both the Intel and Cray developers and this would have not been possible within the context of the full CESM application. Because of the structure of the RRTMG calculation, it was possible to create a number of sub-kernels that were then distributed to multiple developers including a number of graduate students. Using similar optimization techniques to enable vectorization, the cost of the RRTMG radiation calculation was reduced by 40%.

5 Related Work

Computational kernels have been identified as useful tools for analyzing performance of scientific software. Initial simple synthetic computational kernels like Livermore loops [17], and STREAMS [16] led to the development of more complex mini-app. Mini-apps are more complex than computational kernels and may test multiple computer components. Heroex [8] proposed a new role for the mini-app as a tool for, amongst others, studying compiler tuning, and network scaling. Akel [1] examined the viability of using isolated codelets, or kernels, in the place of the original application for performance characterization and optimization. Barret [3] introduced a methodology for assessing the ability of miniapps to effectively represent performance characteristics of original full application.

Heroex initiated a Mini-app aggregation project called Mantevo [8]. Similar projects to share open-source packages followed. The UKMAC consortium [7] was formed amongst several institutions to better understand algorithms and future technologies through compute kernels. The RIKEN Advanced Institute for Computational Science has developed and maintained a suite of mini-apps called FIBER [15].

Lee and Hall [13] described automated kernel extraction using a tool called "Code Isolator". The tool is based on Stanford SUIF compiler and generates a stand-alone source code with representative input state. Liao[14] developed "ROSE outliner" that extracts tunable codes out of a target application. The goal of the tool is to convert whole program tuning problem into a set of manageable multiple kernel tuning problems. "Codelet Finder" from CAPS Enterprise detects hotspot and generates a codelet, kernel, with entire memory state. While previous tools are based on static source analysis, CERE [5] developed by Oliveira works on Intermittent Representation of LLVM compiler suite. It extracts a kernel in the format of IR with page-based memory state. Unlike previous work on code extraction, KGEN only requires Python 2.6, and generates portable standalone executables.

6 Conclusion

In this paper, we introduced a Python-based tool, called KGEN that extracts a part of Fortran code from an application. The tool also provides a simple way to generate state data to drive and verify the execution of the extracted kernel. To extract a kernel from a large application, it supports extraction from MPI applications, application of preprocessing, and limiting searches by excluding identifiers from kernel extraction. We have utilized KGEN to extract thirty computational kernels from a large Fortran based climate simulation application. These kernels are extremely useful in simplifying work on performance optimization, application verification, and benchmarking. For future work, we plan to introduce dynamic analysis into the tool for performance characterization of the kernel. This will help to generate a set of state data that will better represent the performance characteristics of original application.

7 Acknowledgments

This work was supported in part by Intel Parallel Computing Center focused on Weather and Climate Simulation (IPCC-WACS).

References

[1] C. Akel, Y. Kashnikov, P. d. O. Castro, and W. Jalby. Is Source-code Isolation Viable for Performance Characterization? *Parallel Processing (ICPP), 2013 42nd International Conference*, pages 977–984, 2013.

[2] A. H. Baker, D. M. Hammerling, M. N. Levy, H. Xu, J. M. Dennis, B. E. Eaton, J. Edwards, C. Hannay, S. A. Mickelson, R. B. Neale, D. Nychka, J. Shollenberger, J. Tribbia, M. Vertenstein, and D. Williamson. A New Ensemble-Based Consistency Test for the Community Earth System Model. *Geoscientific Model Development Discussions*, 8(5):3823–3859, 2015.

[3] R. F. Barrett, P. S. Crozier, D. W. Doerfler, M. A. Heroux, P. T. Lin, H. K. Thornquist, T. G. Trucano, and C. T. Vaughan. Assessing the role of mini-applications in predicting key performance characteristics of scientific and engineering applications. *Journal of Parallel and Distributed Computing*, 75:107 – 122, 2015.

[4] S. A. Clough, M. W. Shephard, E. J. Mlawer, J. S. Delamere, M. J. Iacono, K. Cady-Pereira, S. Boukabara, and P. D. Brown. Atmospheric Radiative Transfer Modeling: A Summary of the AER Codes. *Journal of Quantitative Spectroscopy and Radiative Transfer*, 91(2):233 – 244, 2005.

[5] P. d. O. Castro, C. Akel, E. Petit, M. Popov, and W. Jalby. CERE: LLVM-Based Codelet Extractor and REplayer for Piecewise Benchmarking and Optimization. *ACM Trans. Archit. Code Optim.*, 12(1):6:1–6:24, April 2015.

[6] A. Gettelman and H. Morrison. Advanced Two-Moment Bulk Microphysics for Global Models. Part I: Off-Line Tests and Comparison with Other Schemes. *J. of Climate*, 28(3):1268–1287, 2015.

[7] J. A. Herdman, W. P. Gaudin, A. C. Mallinson, et al. UK Mini-App Consortium. March 30, 2016. http://uk-mac.github.io.

[8] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving Performance via Mini-Applications. *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, 2009.

[9] J. W. Hurrell et al. The Community Earth System Model: A Framework for Collaborative Research. *Bulletin of the American Meteorological Society*, 94:1339–1360, September 2013.

[10] C. Janssen, D. Quinlan, and J. Shalf. Architectural Simulation for Exascale Hardware/Software Co-Design. Technical report, Sandia National Laboratories (SNL-CA), Livermore, CA (United States), 2011.

[11] Y. Kim, J. M. Dennis, R. R. Kumar, and A. Simha. Fortran Kernel Generator(KGEN) on Github, March 30, 2016. https://github.com/NCAR/KGen.

[12] Y. Kim, J. M. Dennis, R. R. Kumar, and A. Simha. KGEN-Generated Kernels on Github. March 30, 2016. https://github.com/NCAR/kernelOptimization.

[13] Y. Lee and M. Hall. A Code Isolator: Isolating Code Fragments from Large Programs. *Languages and Compilers for High Performance Computing*, 3602:164–178, 2005.

[14] C. Liao, D. J. Quinlan, R. Vuduc, and T. Panas. Effective Source-to-Source Outlining to Support Whole Program Empirical Optimization. *Languages and Compilers for Parallel Computing*, 5898:308–322, 2010.

[15] N. Maruyama, S. Suzuki, et al. Fiber Miniapp Suite, July 23, 2015. http://fiber-miniapp.github.io.

[16] J. D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[17] F. H. McMahon. The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, Livermore, California, December 1986.

[18] H. Morrison and A. Gettelman. A New Two-Moment Bulk Stratiform Cloud Microphysics Scheme in the Community Atmosphere Model, Version 3 (CAM3). Part I: Description and Numerical

Tests. *J. of Climate*, 21(15):3642–3659, 2008.

[19] R. Rew and G. Davis. NetCDF: An Interface for Scientific Data Access. *Computer Graphics and Applications, IEEE*, 10(4):76–82, July 1990.

[20] W. C. Skamarock, S. Park, J. B. Klemp, and C. Snyder. Atmospheric Kinetic Energy Spectra from Global High-Resolution Nonhydrostatic Simulations. *J. Atmos. Sci.*, 71:4369–4381, 2014.